
"Wunschkonzert"
Grundlagen der Programmiersprachen 2011

Dr. Bastian Cramer

Wünsche

- ▶ Linearisierung
- ▶ Laufzeitkeller
- ▶ Funktionale Programmierung
 - ▶ Grundbegriffe
 - ▶ Funktionen höherer Ordnung

Linearisierte Speicherung von Arrays

a: array[u₁..o₁, u₂..o₂, ..., u_n..o_n] of int;

Untergrenze, Obergrenze, Ebene

Linearisierte Speicherung von Arrays

a: array[u₁..o₁, u₂..o₂, ..., u_n..o_n] of int;

Untergrenze, Obergrenze, Ebene

Elementgröße, z.B. 4 Byte

Linearisierte Speicherung von Arrays

```
a: array[u1..o1, u2..o2, ..., un..on] of int;
```

Untergrenze, Obergrenze, Ebene

Elementgröße, z.B. 4 Byte

```
a: array[21..31, 42..52, 63..73] of int;
```

Linearisierte Speicherung von Arrays

a: array[u₁..o₁, u₂..o₂, ..., u_n..o_n] of int;

Untergrenze, Obergrenze, Ebene

Elementgröße, z.B. 4 Byte

a: array[2₁..3₁, 4₂..5₂, 6₃..7₃] of int;

1 Element = 4 Byte



Linearisierte Speicherung von Arrays

a: array[u₁..o₁, u₂..o₂, ..., u_n..o_n] of int;

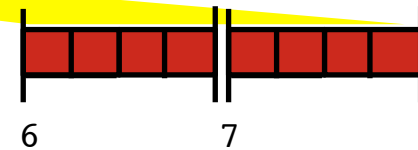
Untergrenze, Obergrenze, Ebene

Elementgröße, z.B. 4 Byte

a: array[

6₃..7₃] of int;

| Element = 4 Byte



Linearisierte Speicherung von Arrays

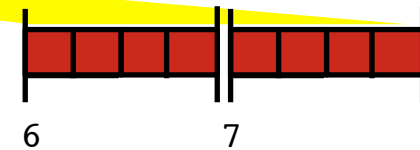
a: array[$u_1 \dots o_1$, $u_2 \dots o_2$, ..., $u_n \dots o_n$] of int;

Untergrenze, Obergrenze, Ebene

Elementgröße, z.B. 4 Byte

a: array[$4_2 \dots 5_2$, $6_3 \dots 7_3$] of int;

1 Element = 4 Byte



$$5-4+1 = 2 \times \text{Ebene } 3$$

Linearisierte Speicherung von Arrays

a: array[u₁..o₁, u₂..o₂, ..., u_n..o_n] of int;

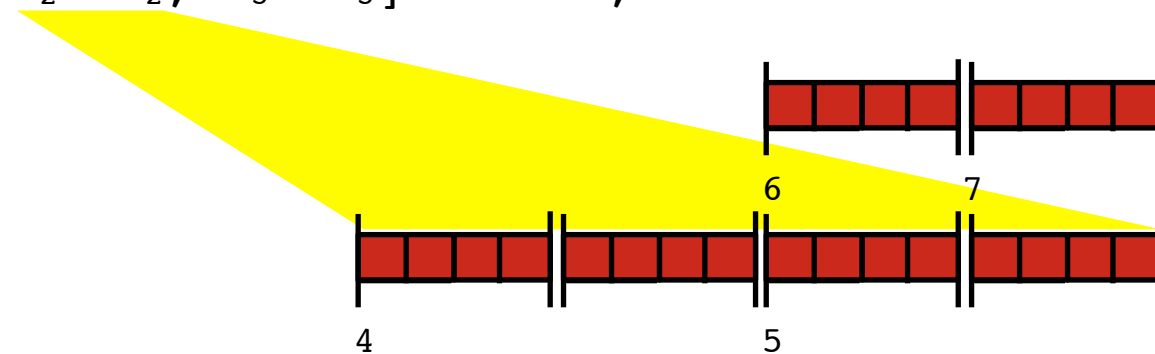
Untergrenze, Obergrenze, Ebene

Elementgröße, z.B. 4 Byte

a: array[4₂..5₂, 6₃..7₃] of int;

1 Element = 4 Byte

5-4+1 = 2 x Ebene 3



Linearisierte Speicherung von Arrays

a: array[$u_1 \dots o_1$, $u_2 \dots o_2$, ..., $u_n \dots o_n$] of int;

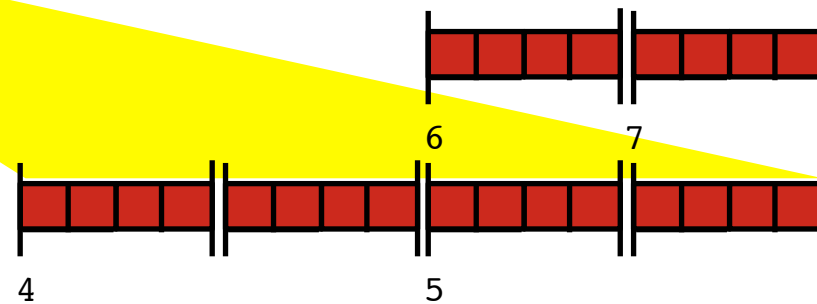
Untergrenze, Obergrenze, Ebene

Elementgröße, z.B. 4 Byte

a: array[$4_2 \dots 5_2$, $6_3 \dots 7_3$] of int;

1 Element = 4 Byte

$5 - 4 + 1 = 2 \times$ Ebene 3



$$\text{store}[j, k] = \text{store}[5, 6] = (j - u_j) * (o_k - u_k + 1) * 4$$

Linearisierte Speicherung von Arrays

a: array[$u_1 \dots o_1$, $u_2 \dots o_2$, ..., $u_n \dots o_n$] of int;

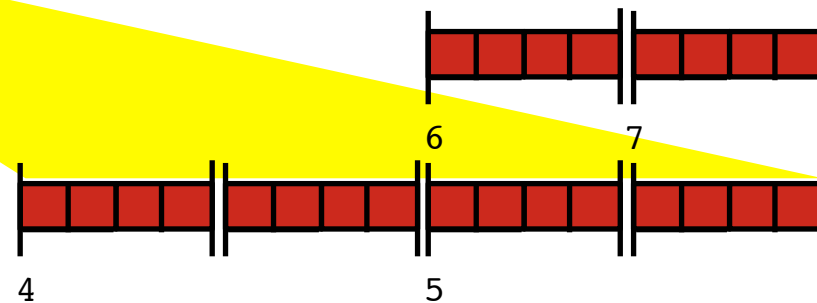
Untergrenze, Obergrenze, Ebene

Elementgröße, z.B. 4 Byte

a: array[$4_2 \dots 5_2$, $6_3 \dots 7_3$] of int;

1 Element = 4 Byte

$5 - 4 + 1 = 2 \times$ Ebene 3



$$\text{store}[j, k] = \text{store}[5, 6] = (j - u_j) * (o_k - u_k + 1) * 4$$

Linearisierte Speicherung von Arrays

a: array[$u_1 \dots o_1$, $u_2 \dots o_2$, ..., $u_n \dots o_n$] of int;

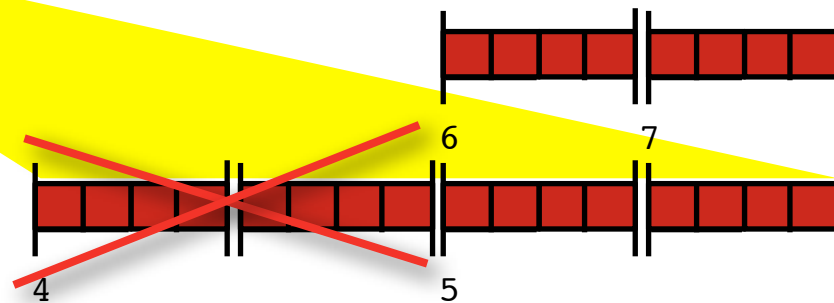
Untergrenze, Obergrenze, Ebene

Elementgröße, z.B. 4 Byte

a: array[$4_2 \dots 5_2$, $6_3 \dots 7_3$] of int;

1 Element = 4 Byte

$5 - 4 + 1 = 2 \times$ Ebene 3



$$\text{store}[j, k] = \text{store}[5, 6] = \underline{(j - u_j)} * (o_k - u_k + 1) * 4$$

n j-Ebenen
überspringen

Linearisierte Speicherung von Arrays

a: array[$u_1 \dots o_1$, $u_2 \dots o_2$, ..., $u_n \dots o_n$] of int;

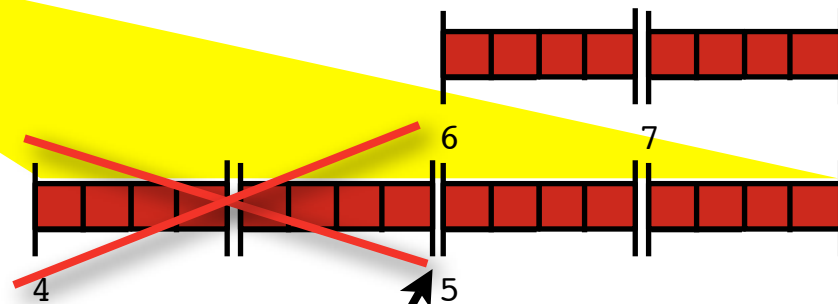
Untergrenze, Obergrenze, Ebene

Elementgröße, z.B. 4 Byte

a: array[$4_2 \dots 5_2$, $6_3 \dots 7_3$] of int;

1 Element = 4 Byte

$5 - 4 + 1 = 2 \times$ Ebene 3



$$\text{store}[j, k] = \text{store}[5, 6] = \underbrace{(j - u_j)}_{n \text{ j-Ebenen überspringen}} * (o_k - u_k + 1) * 4$$

Linearisierte Speicherung von Arrays

a: array[$u_1 \dots o_1$, $u_2 \dots o_2$, ..., $u_n \dots o_n$] of int;

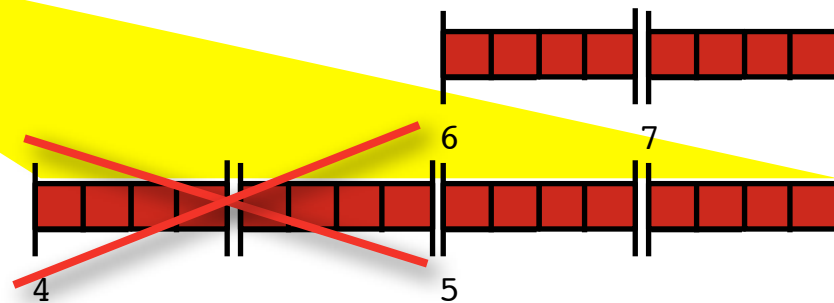
Untergrenze, Obergrenze, Ebene

Elementgröße, z.B. 4 Byte

a: array[$4_2 \dots 5_2$, $6_3 \dots 7_3$] of int;

1 Element = 4 Byte

$5 - 4 + 1 = 2 \times$ Ebene 3



$$\text{store}[j, k] = \text{store}[5, 6] = \underbrace{(j - u_j)}_{n \text{ j-Ebenen überspringen}} * \underbrace{(o_k - u_k + 1) * 4}_{\text{multiplizieren mit allen nachfolgenden Ebenen}}$$

n j-Ebenen
überspringen

multiplizieren mit
allen nachfolgenden Ebenen

Linearisierte Speicherung von Arrays

a: array[$u_1 \dots o_1$, $u_2 \dots o_2$, ..., $u_n \dots o_n$] of int;

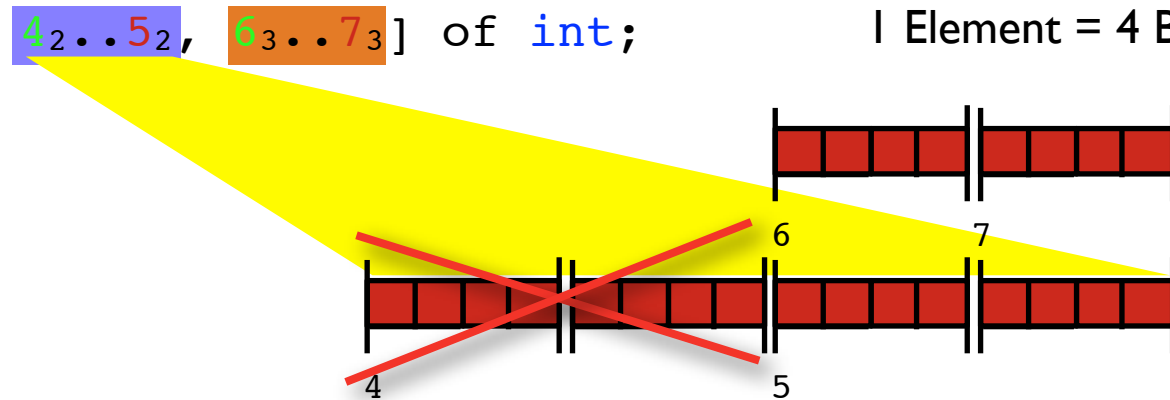
Untergrenze, Obergrenze, Ebene

Elementgröße, z.B. 4 Byte

a: array[$4_2 \dots 5_2$, $6_3 \dots 7_3$] of int;

1 Element = 4 Byte

$5 - 4 + 1 = 2 \times$ Ebene 3



$$\text{store}[j, k] = \text{store}[5, 6] = (j - u_j) * (o_k - u_k + 1) * 4 + \underline{(k - u_k) * 4}$$

Offset in Ebene k

Linearisierte Speicherung von Arrays

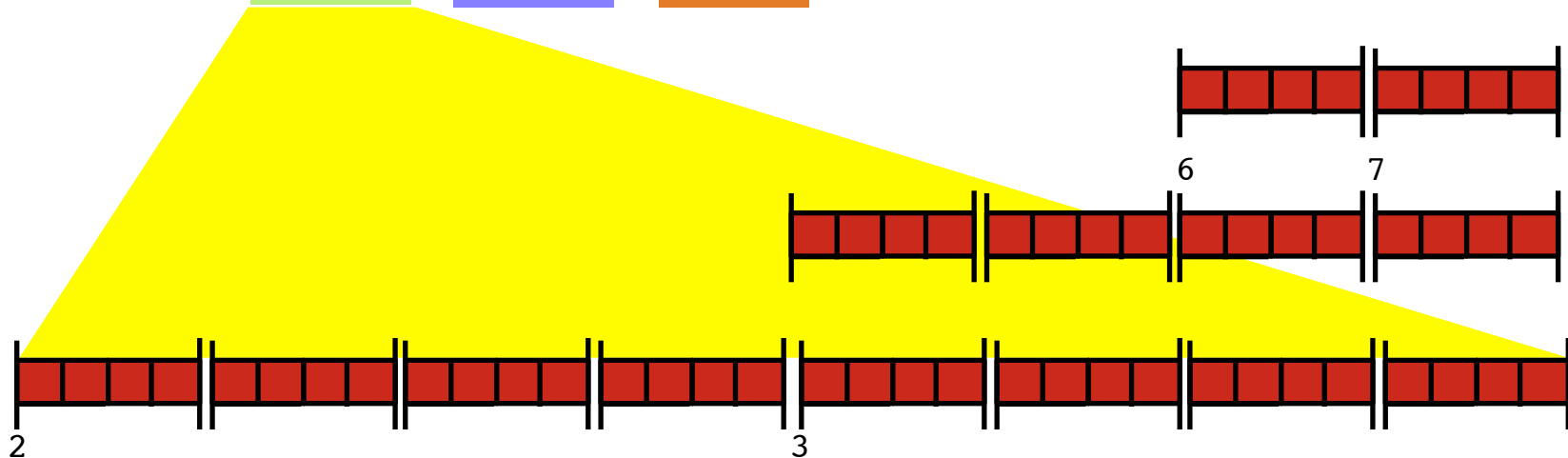
a: array[$u_1 \dots o_1$, $u_2 \dots o_2$, ..., $u_n \dots o_n$] of int;

Untergrenze, Obergrenze, Ebene

Elementgröße, z.B. 4 Byte

a: array[$2_1 \dots 3_1$, $4_2 \dots 5_2$, $6_3 \dots 7_3$] of int;

1 Element = 4 Byte



$3-2+1 = 2 \times$ Ebene 2

store[i, j, k] = store[3, 5, 6] = $(k - u_k) * 4$

+ $(j - u_j) * (o_k - u_k + 1) * 4$

Linearisierte Speicherung von Arrays

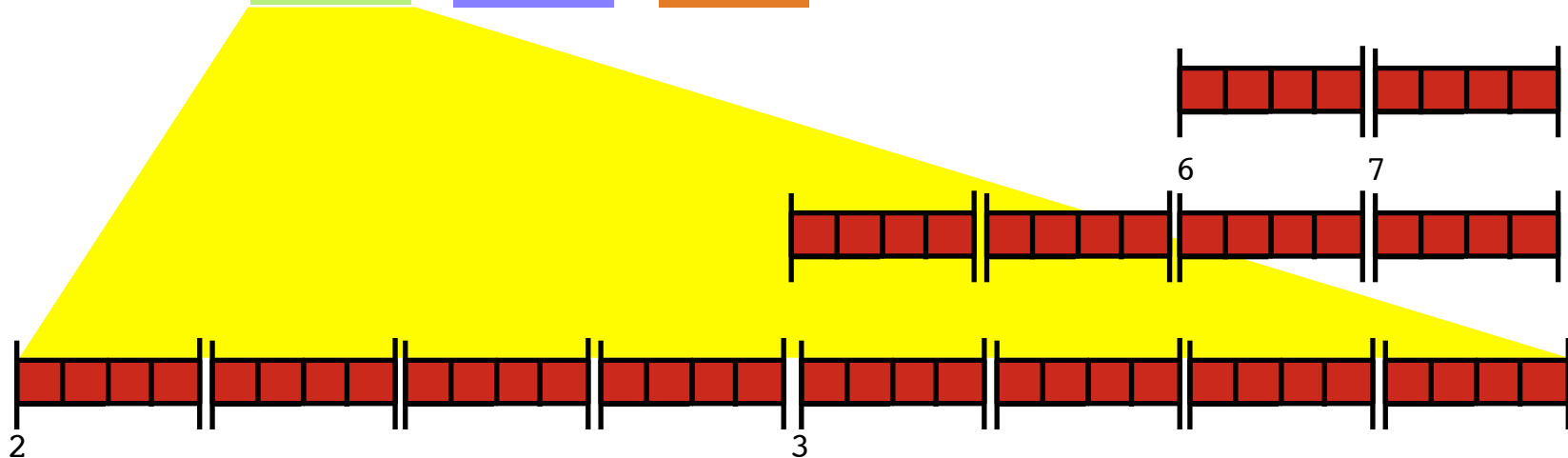
a: array[$u_1 \dots o_1$, $u_2 \dots o_2$, ..., $u_n \dots o_n$] of int;

Untergrenze, Obergrenze, Ebene

Elementgröße, z.B. 4 Byte

a: array[$2_1 \dots 3_1$, $4_2 \dots 5_2$, $6_3 \dots 7_3$] of int;

1 Element = 4 Byte



$3-2+1 = 2 \times \text{Ebene } 2$

store[i, j, k] = store[3, 5, 6] = $(k - u_k) * 4$

$$+ (j - u_j) * (o_k - u_k + 1) * 4$$

$$+ (i - u_i) * (o_j - u_j + 1)$$

Linearisierte Speicherung von Arrays

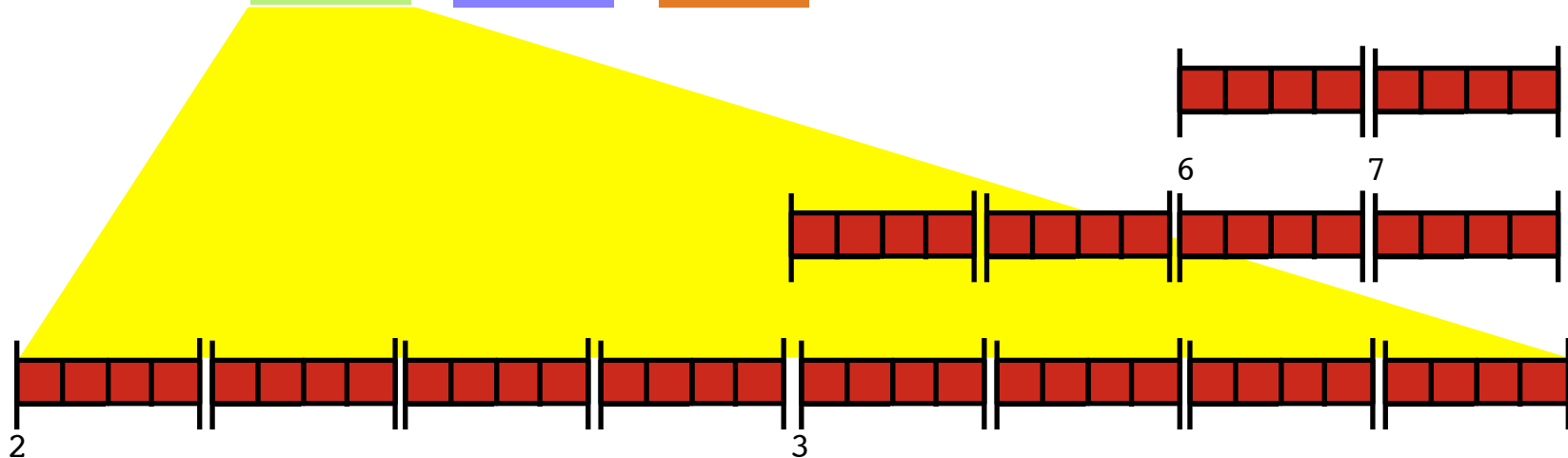
a: array[$u_1 \dots o_1$, $u_2 \dots o_2$, ..., $u_n \dots o_n$] of int;

Untergrenze, Obergrenze, Ebene

Elementgröße, z.B. 4 Byte

a: array[$2_1 \dots 3_1$, $4_2 \dots 5_2$, $6_3 \dots 7_3$] of int;

1 Element = 4 Byte



$3-2+1 = 2 \times$ Ebene 2

store[i, j, k] = store[3, 5, 6] = $(k - u_k) * 4$

$$\begin{aligned}
 &+ (j - u_j) * (o_k - u_k + 1) * 4 \\
 &+ (i - u_i) * (o_j - u_j + 1) * (o_k - u_k + 1)
 \end{aligned}$$

Linearisierte Speicherung von Arrays

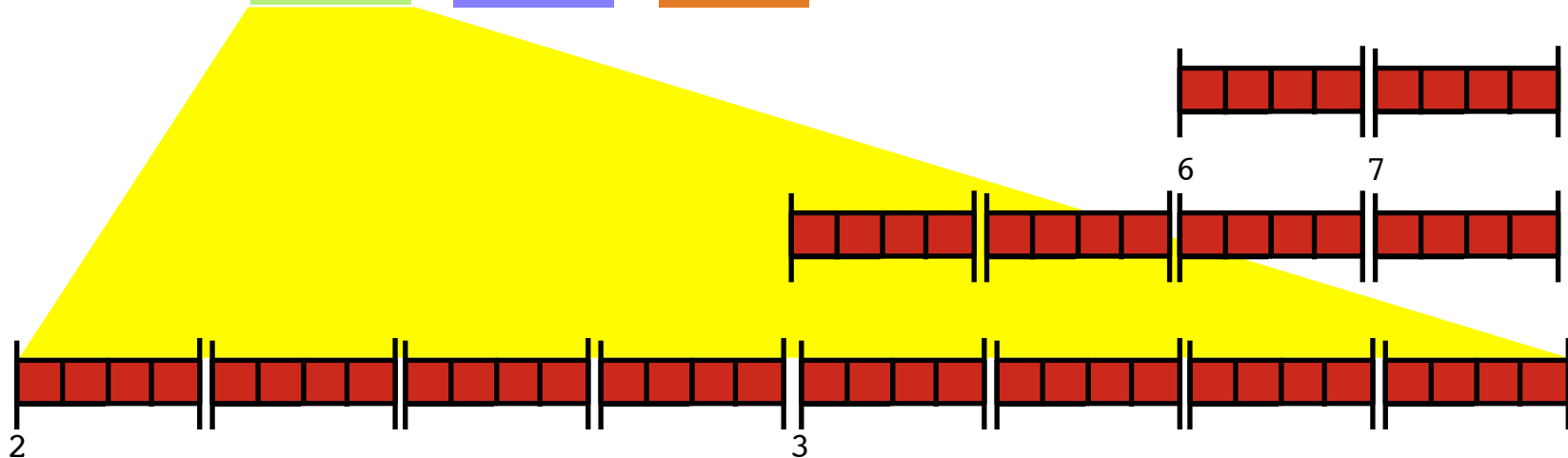
a: array[$u_1 \dots o_1$, $u_2 \dots o_2$, ..., $u_n \dots o_n$] of int;

Untergrenze, Obergrenze, Ebene

Elementgröße, z.B. 4 Byte

a: array[$2_1 \dots 3_1$, $4_2 \dots 5_2$, $6_3 \dots 7_3$] of int;

1 Element = 4 Byte



$3-2+1 = 2 \times$ Ebene 2

store[i, j, k] = store[3, 5, 6] = $(k - u_k) * 4$

+ $(j - u_j) * (o_k - u_k + 1) * 4$

+ $(i - u_i) * (o_j - u_j + 1) * (o_k - u_k + 1) * 4$

+ anf

= $6 - 6 * 4$

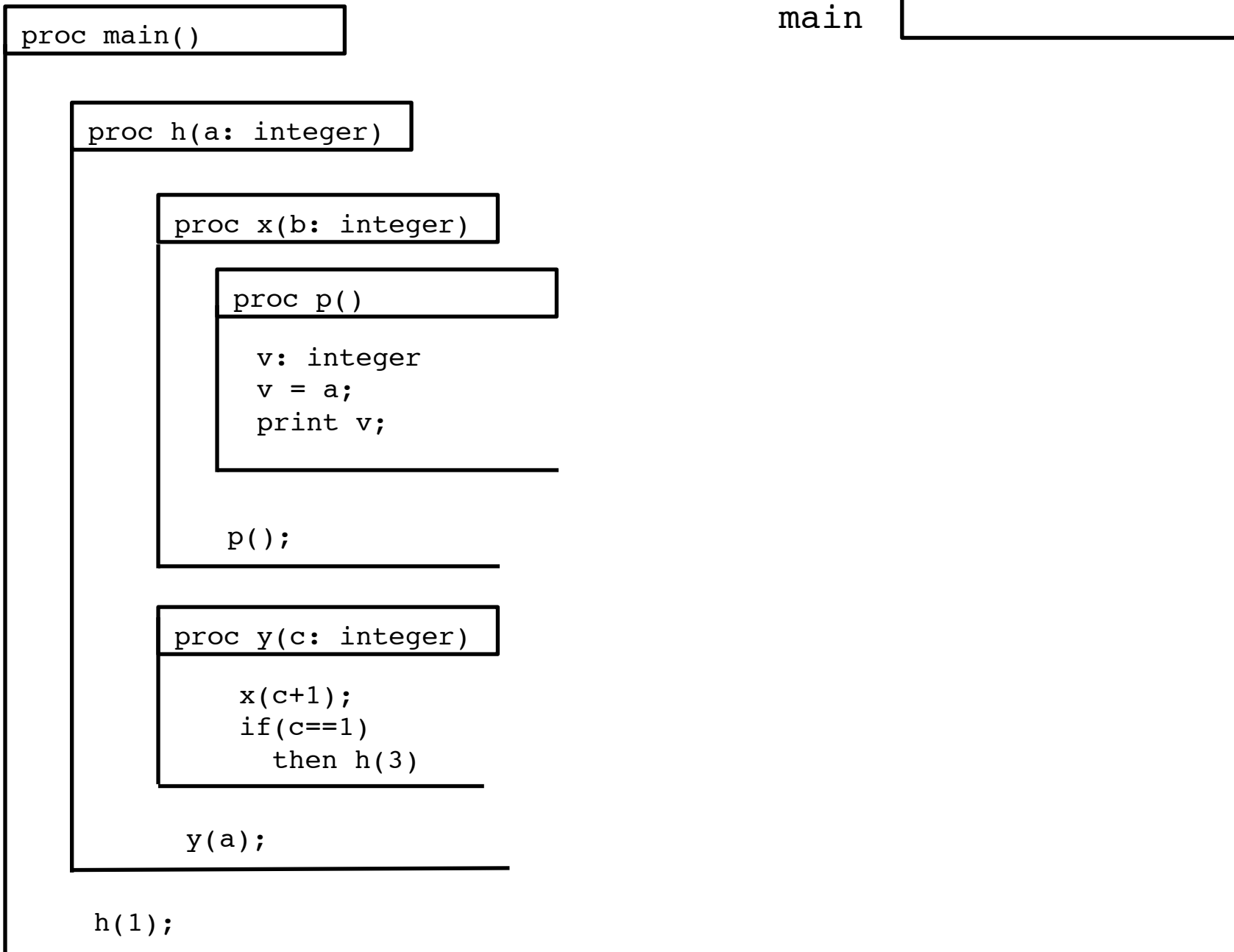
+ $(5 - 4) * (7 - 6 + 1) * 4$

+ $(3 - 2) * (5 - 4 + 1) * (7 - 6 + 1) * 4$

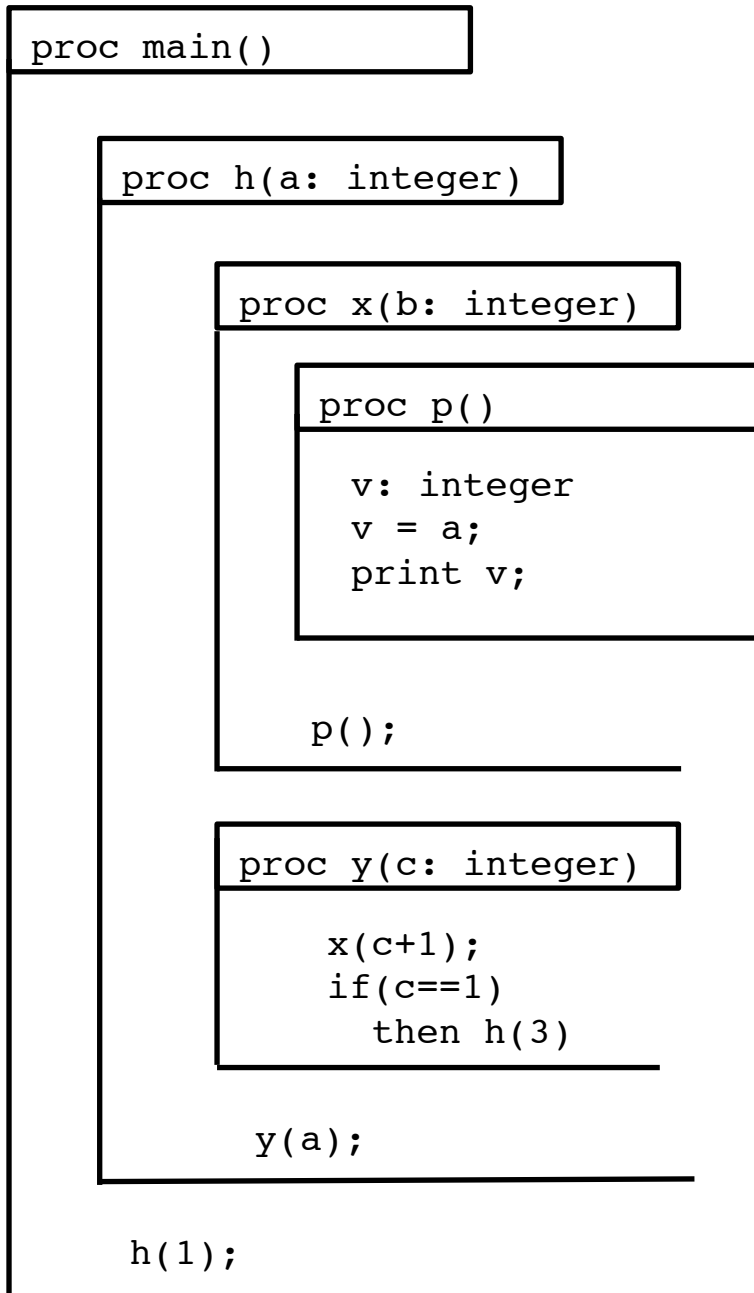
= $8 + 2 * 2 * 4 = 24$

Laufzeitkeller

main

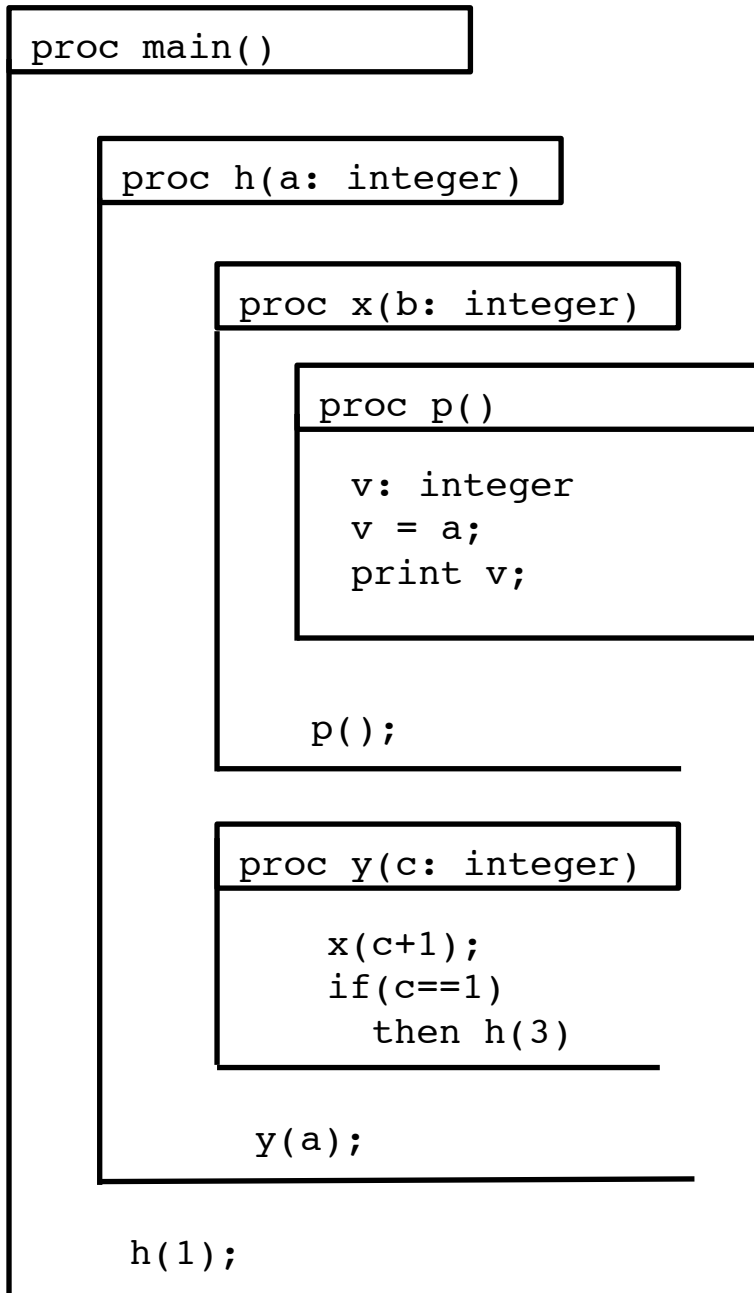


Laufzeitkeller



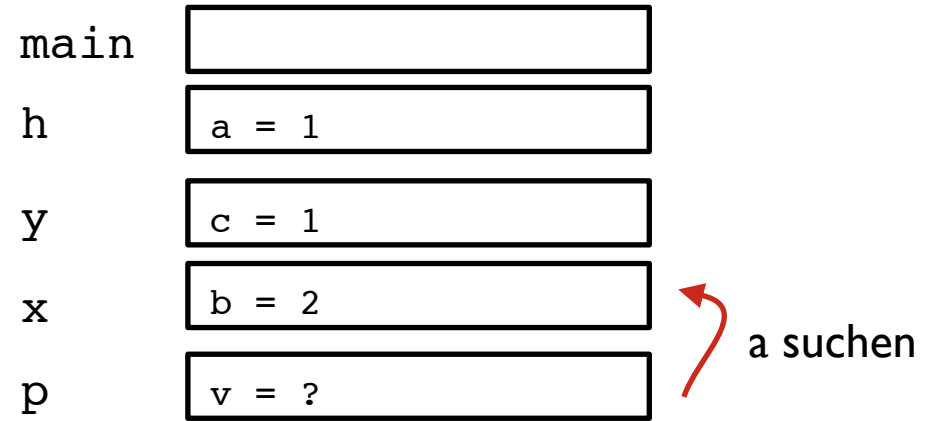
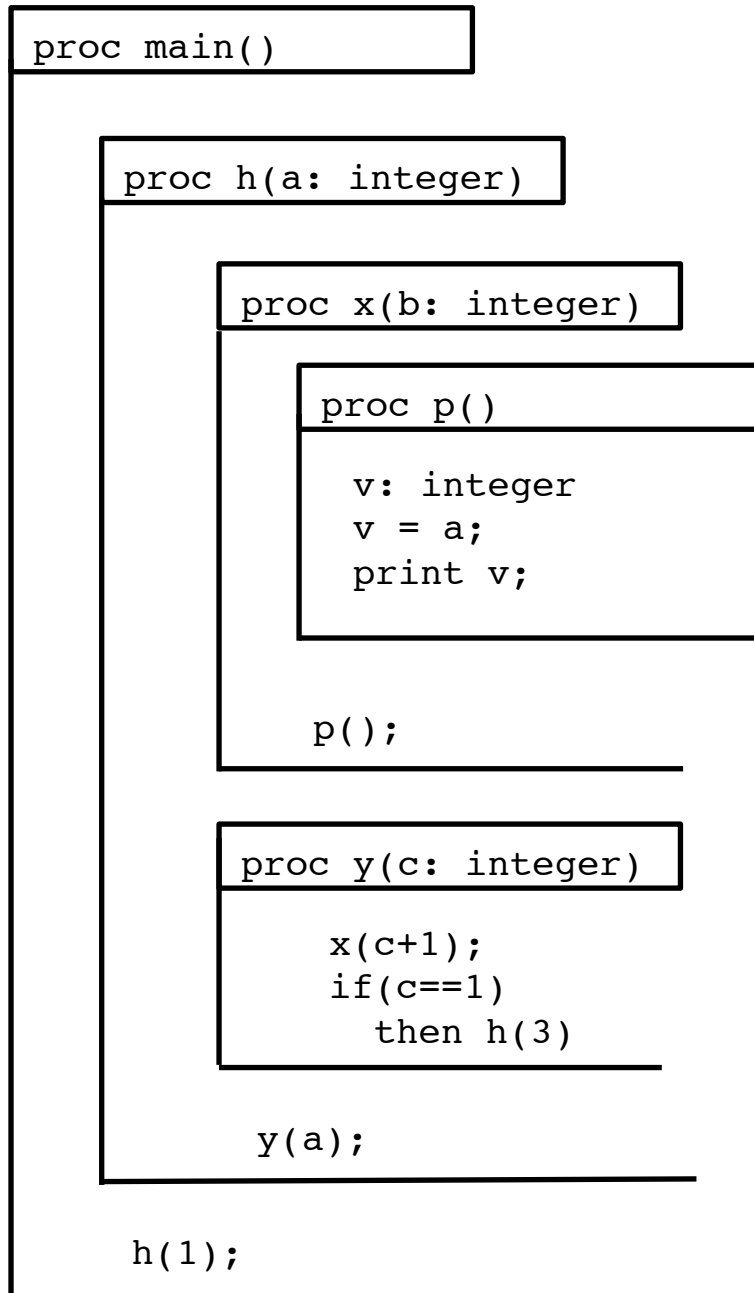
main	
h	a = 1

Laufzeitkeller

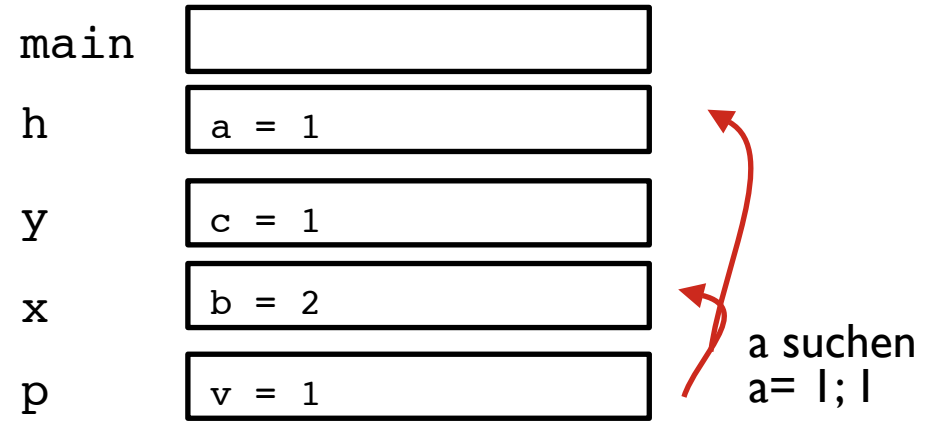
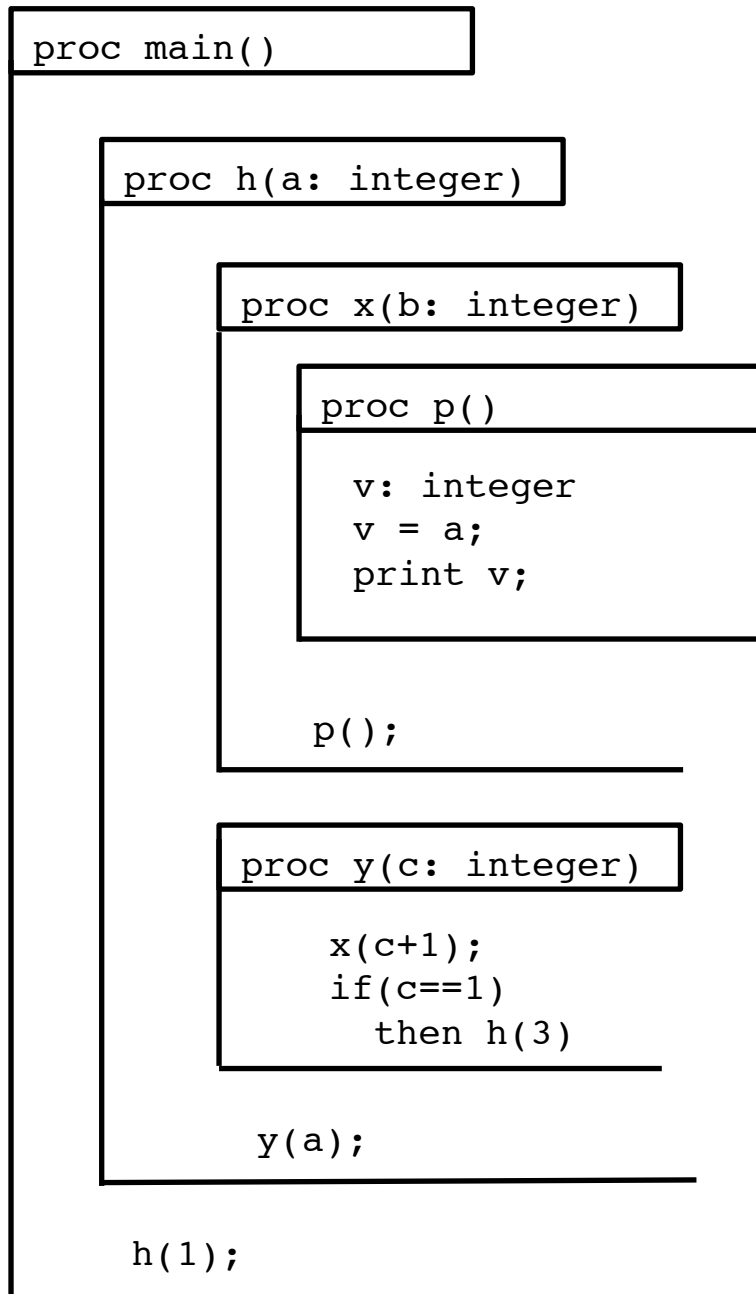


main	
h	a = 1
y	c = 1

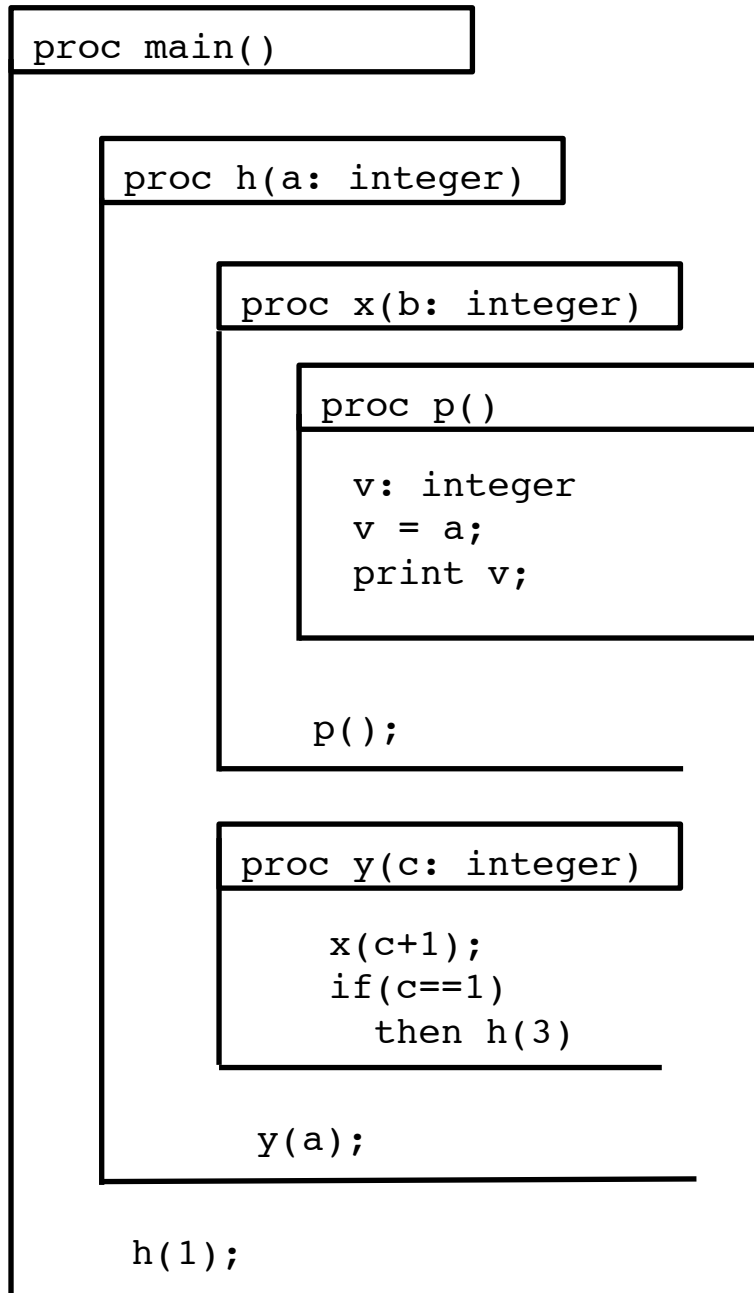
Laufzeitkeller



Laufzeitkeller

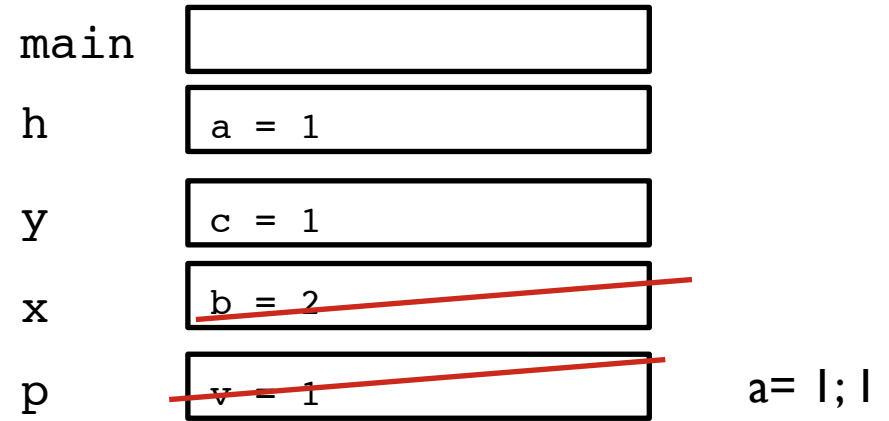
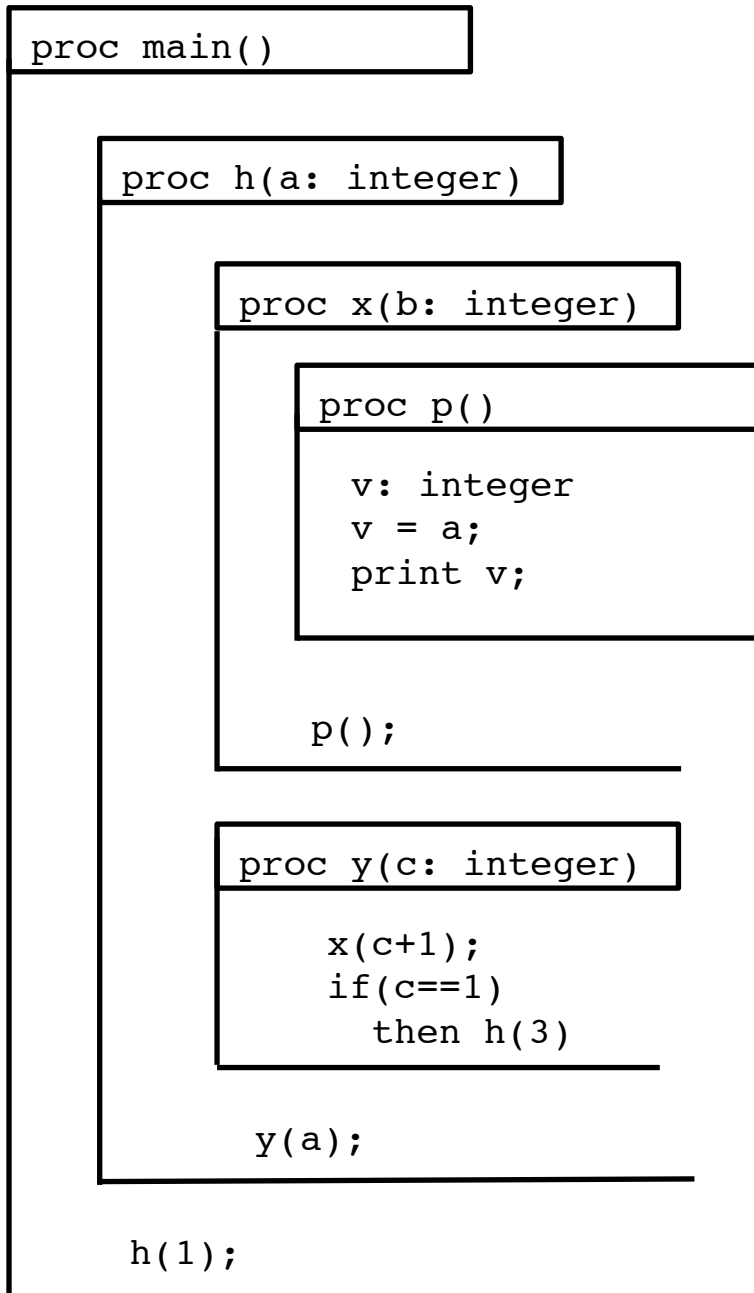


Laufzeitkeller

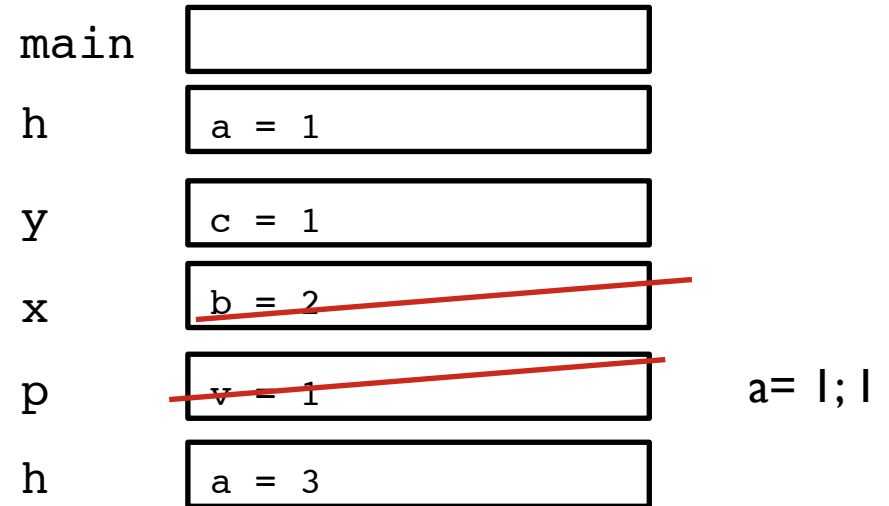
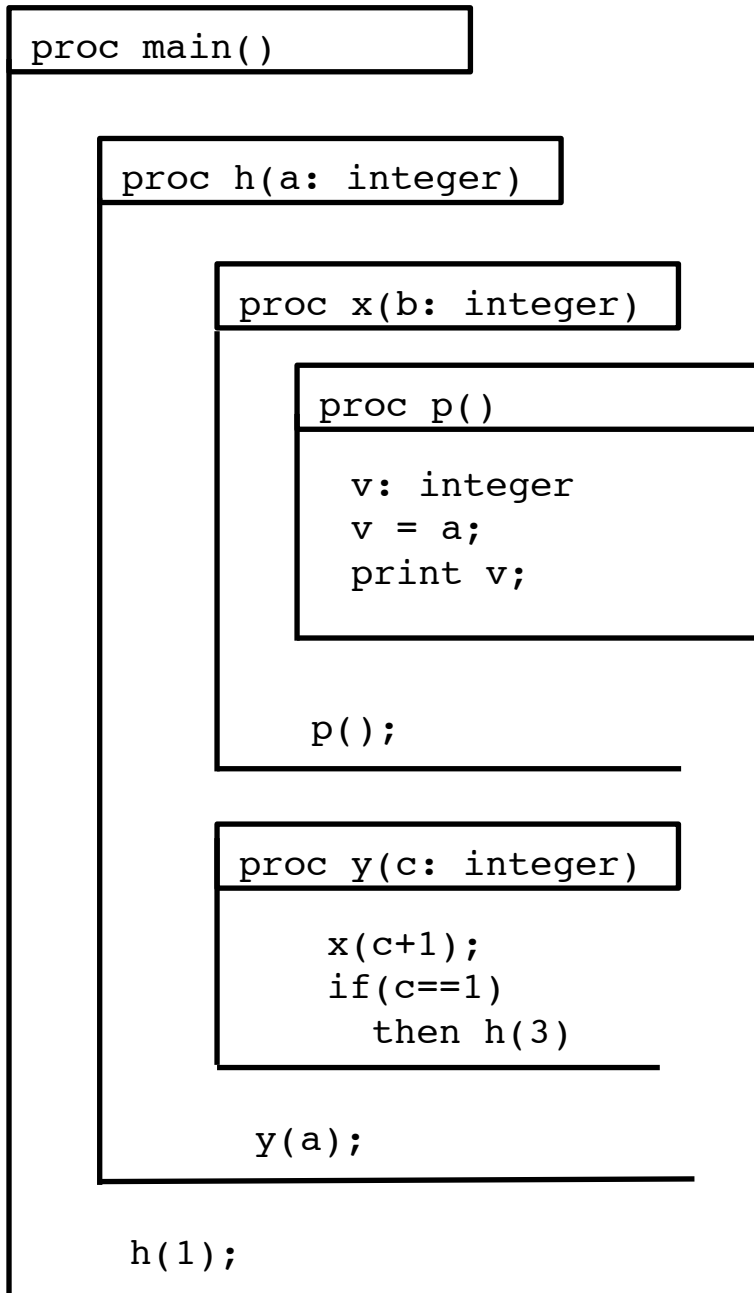


main		
h	a = 1	
y	c = 1	
x	b = 2	
p	v = 1	a = 1; 1

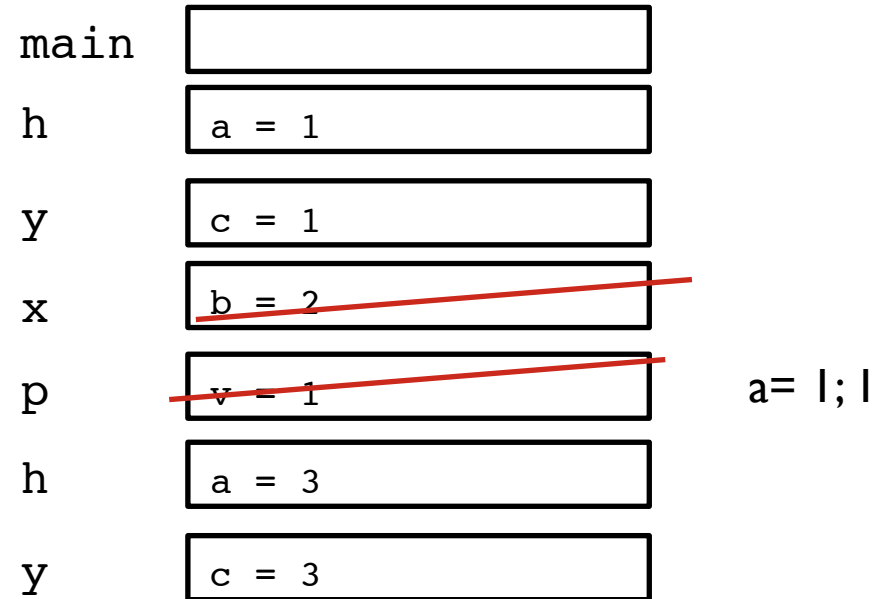
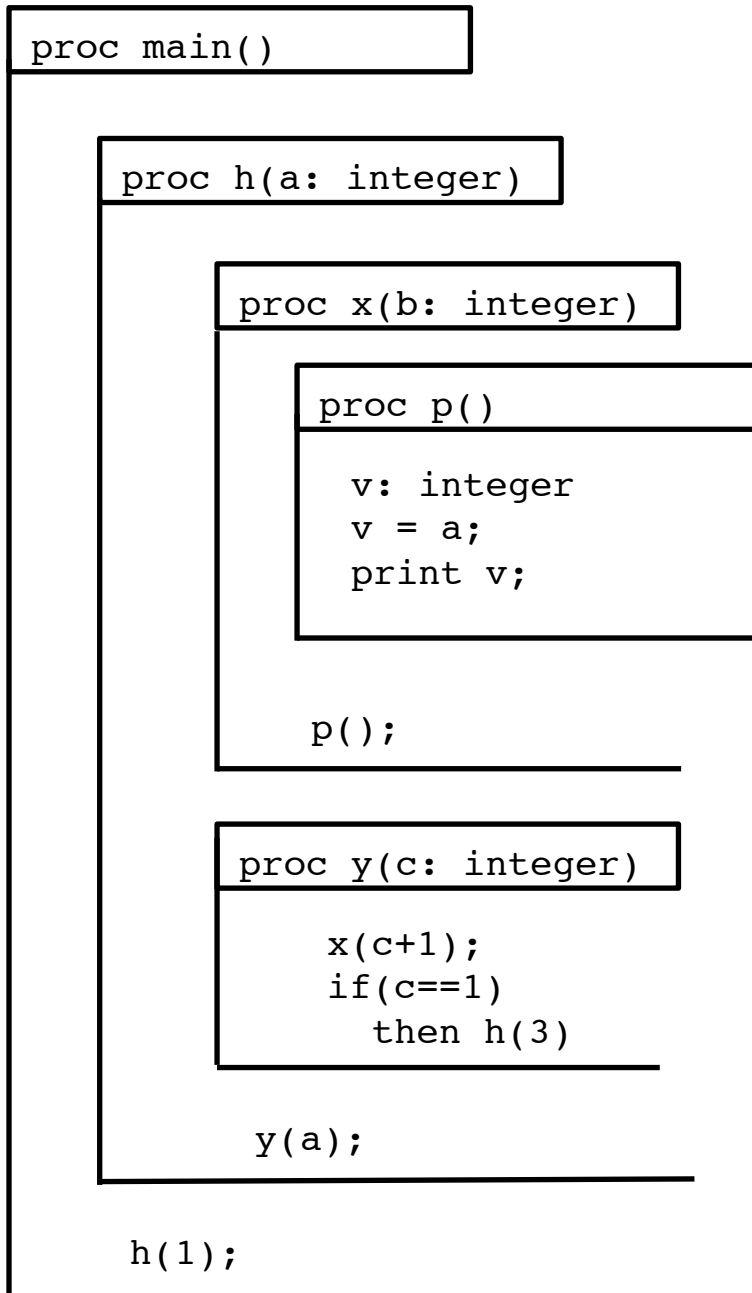
Laufzeitkeller



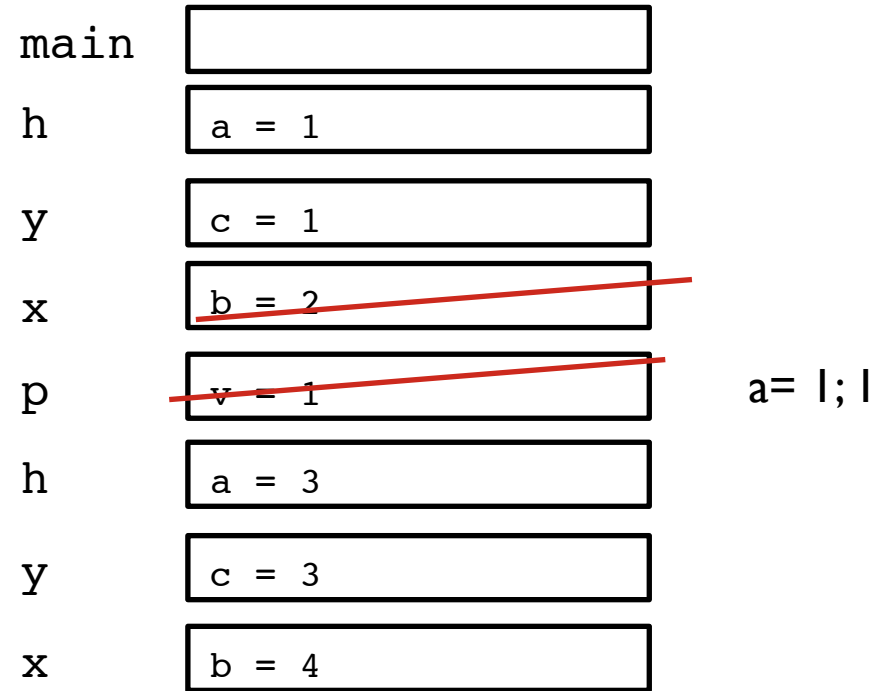
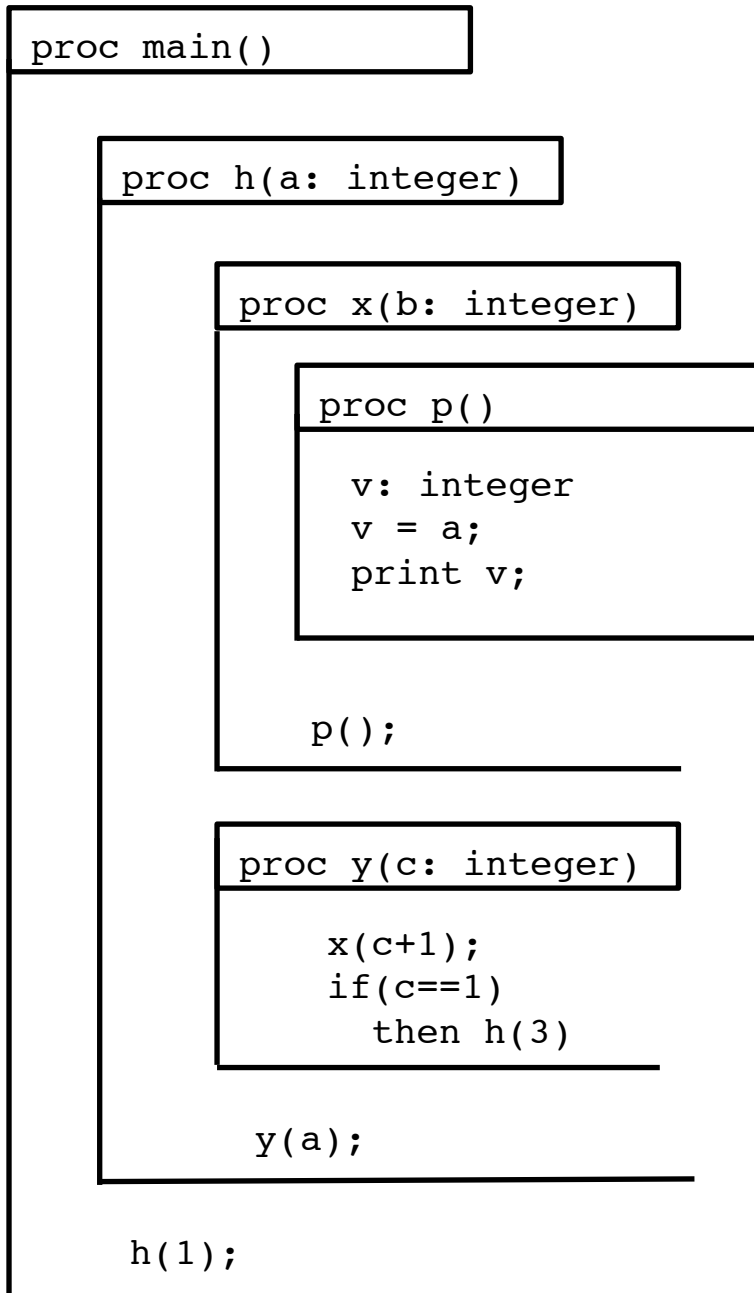
Laufzeitkeller



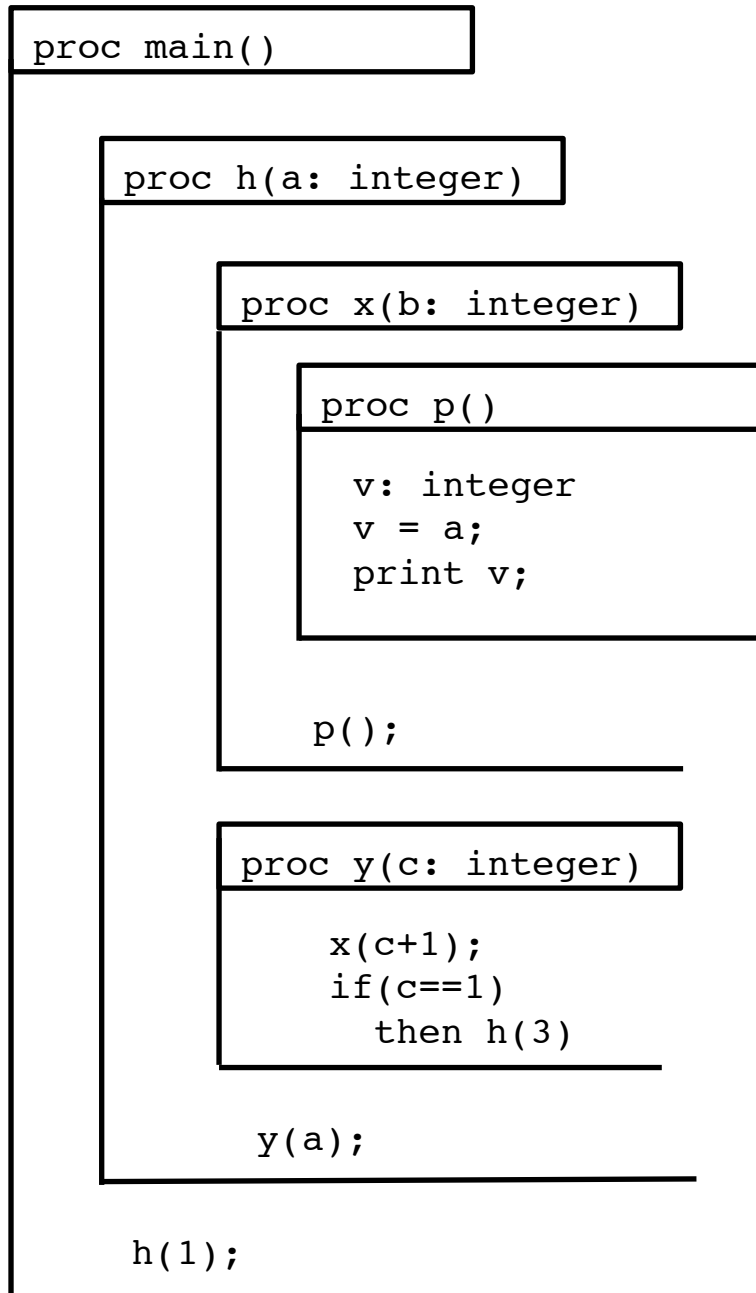
Laufzeitkeller



Laufzeitkeller

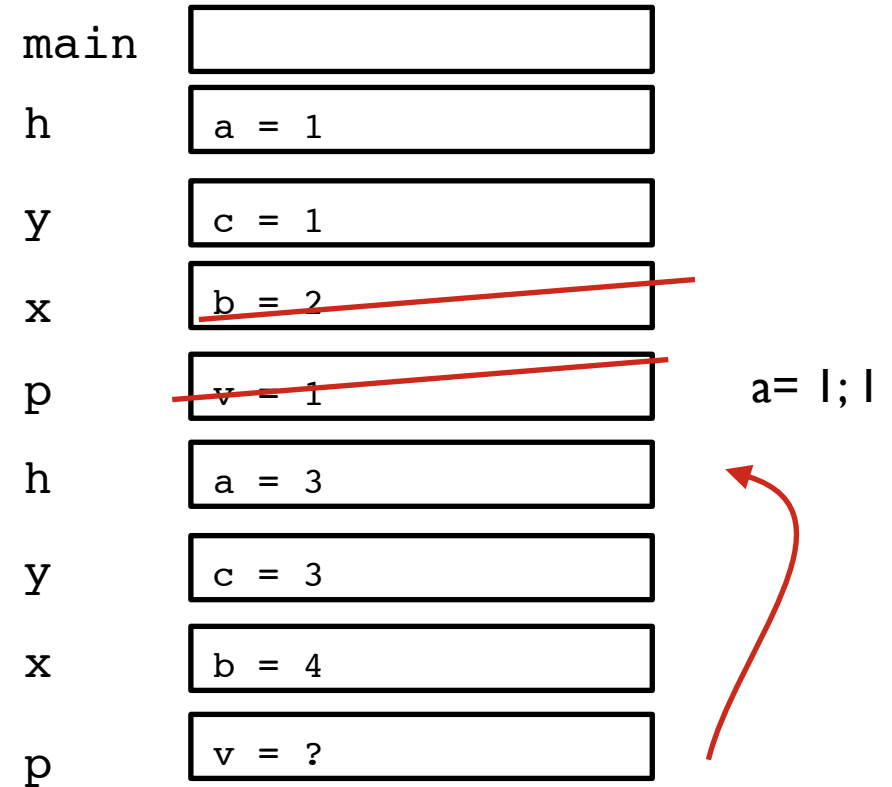
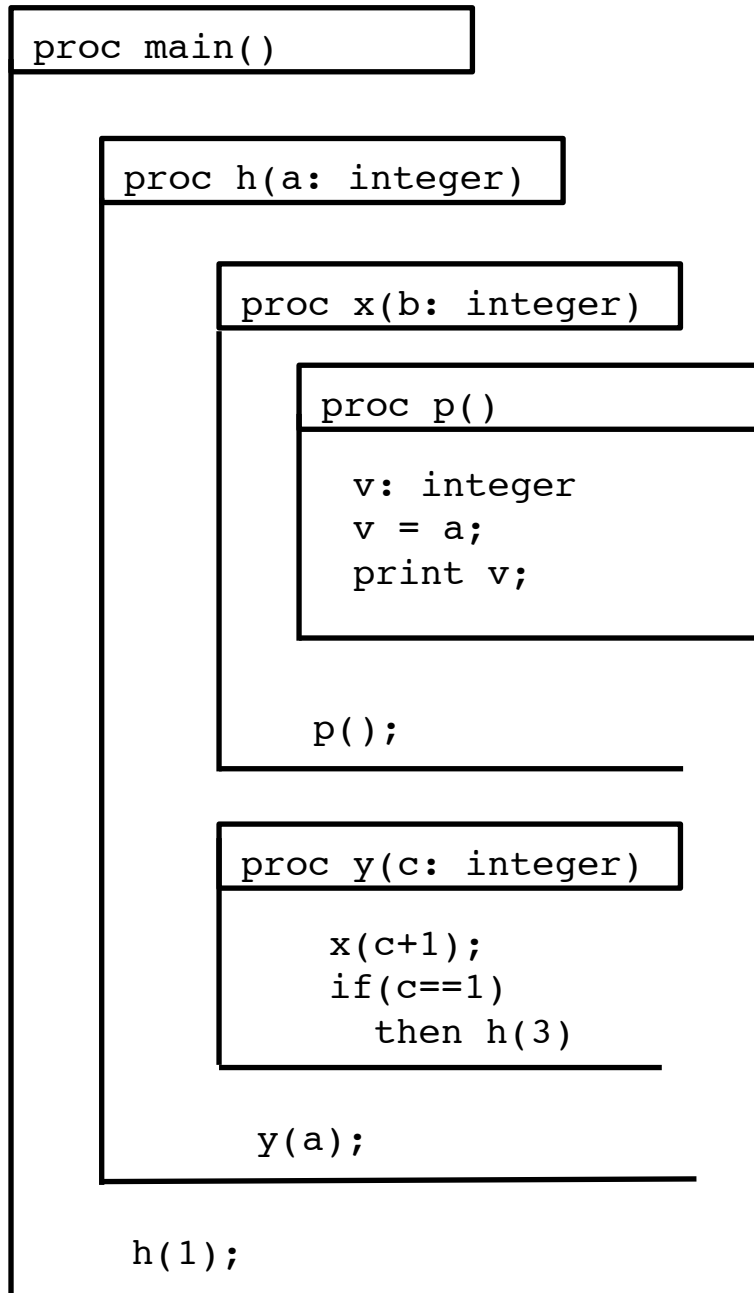


Laufzeitkeller

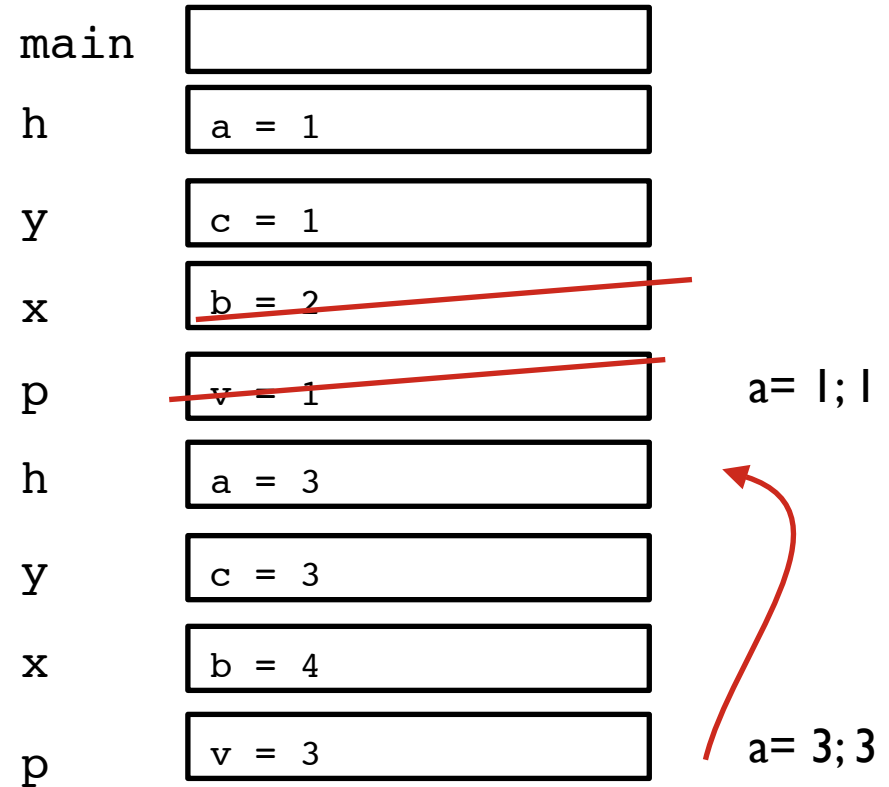
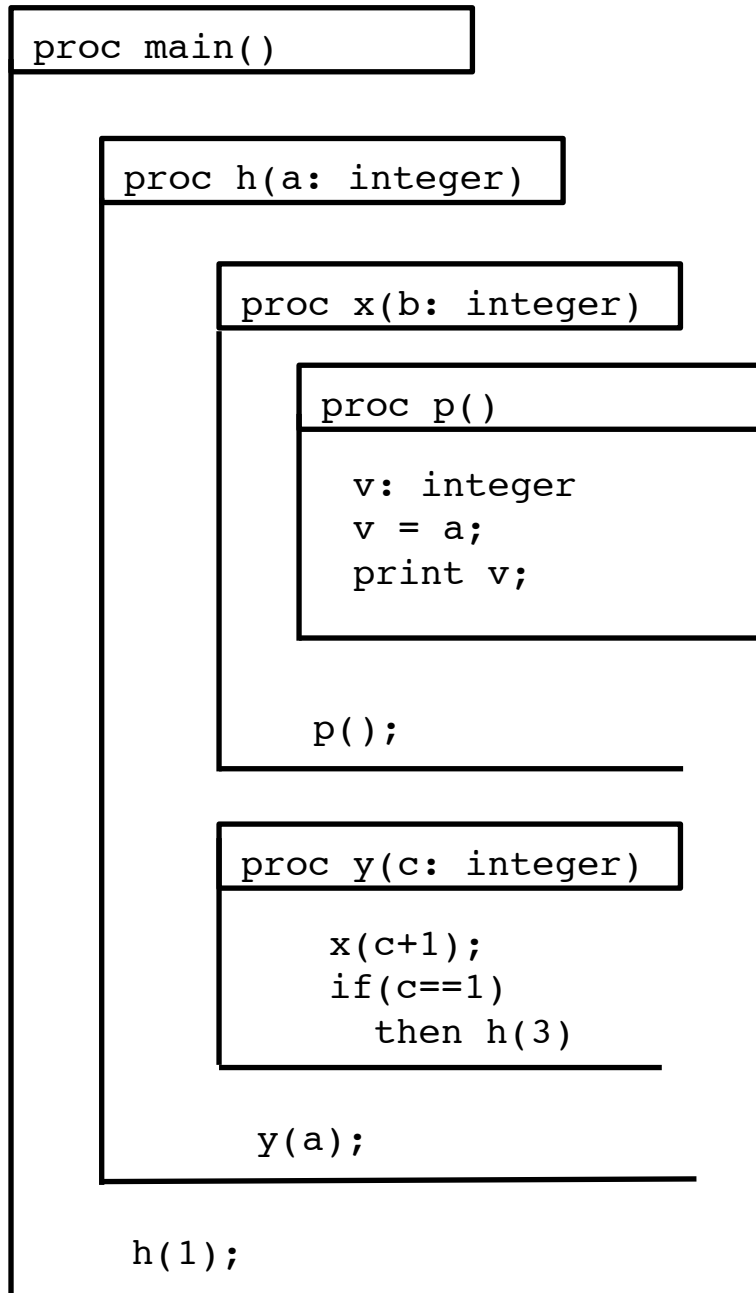


main		
h	a = 1	
y	c = 1	
x	b = 2	
p	v = 1	a = 1; 1
h	a = 3	
y	c = 3	
x	b = 4	
p	v = ?	

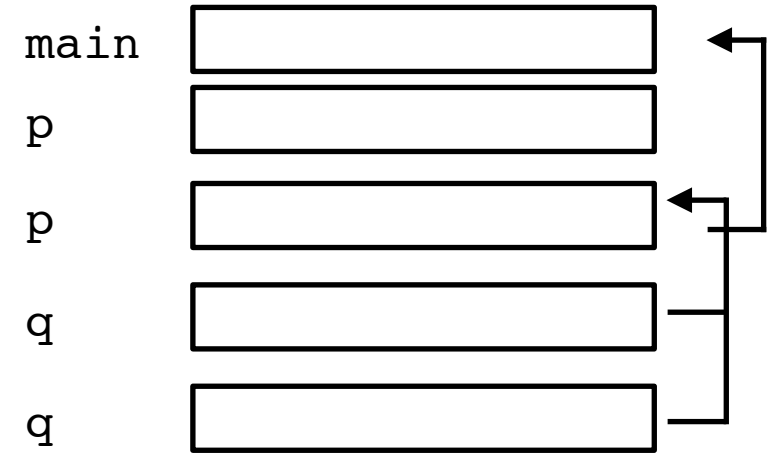
Laufzeitkeller



Laufzeitkeller

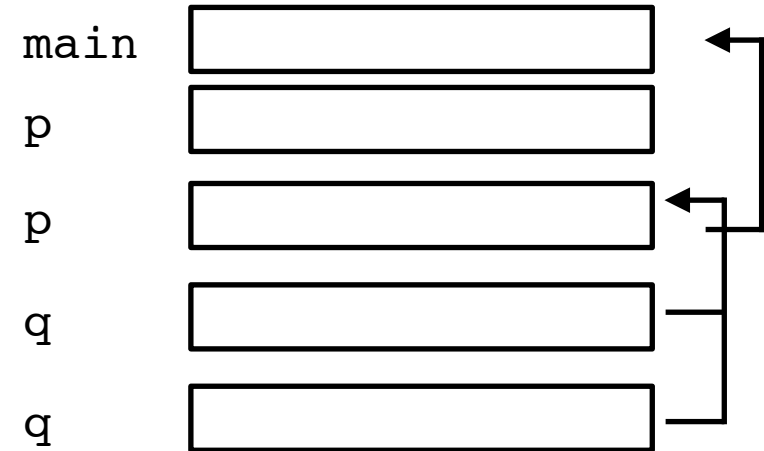


Laufzeitkeller



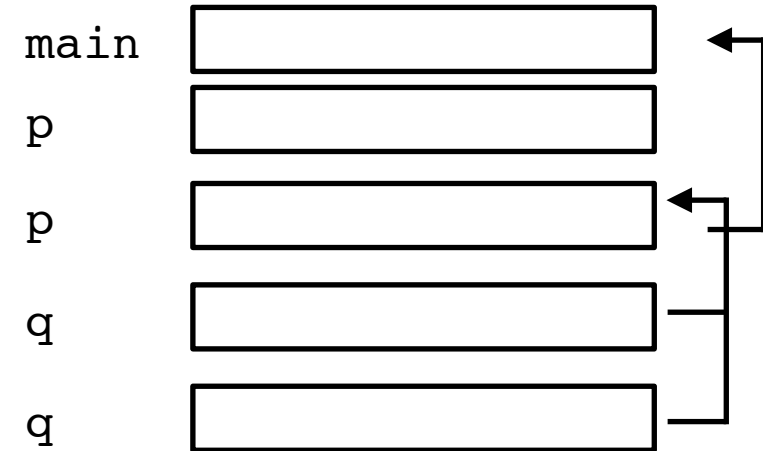
Laufzeitkeller

```
proc main() {  
  proc p( ) {  
  
    proc q( ) {  
  
    }  
  }  
}
```



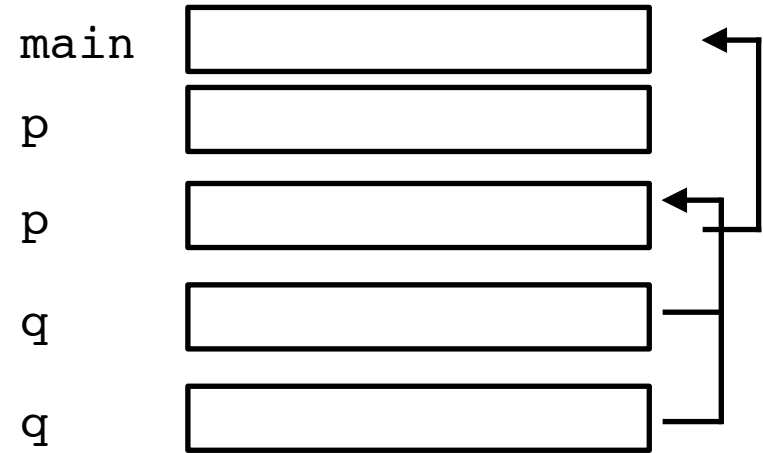
Laufzeitkeller

```
proc main() {  
  proc p(a) {  
  
    proc q( ) {  
  
    }  
  }  
  p(1);  
}
```



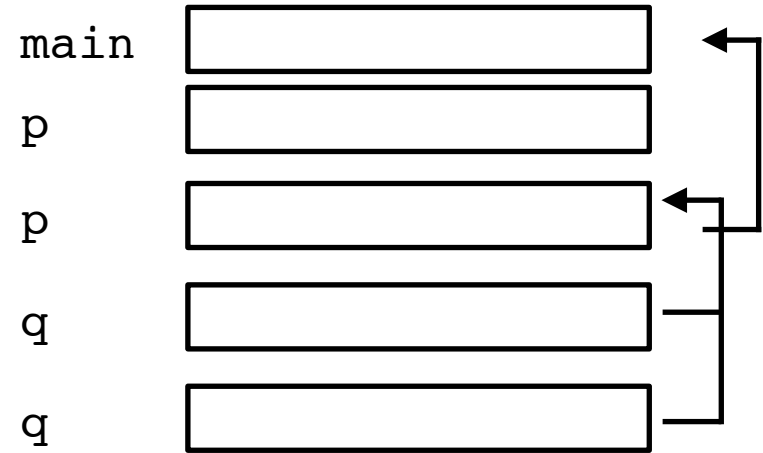
Laufzeitkeller

```
proc main() {  
  proc p(a) {  
    if(a == 1)  
      p(2);  
    else  
      q(1);  
    proc q( ) {  
  
    }  
  }  
  p(1);  
}
```



Laufzeitkeller

```
proc main() {  
  proc p(a) {  
    if(a == 1)  
      p(2);  
    else  
      q(1);  
    proc q(b) {  
      if(b == 1)  
        q(2);  
    }  
  }  
  p(1);  
}
```



Funktionale Programmierung

Grundkonzepte: Funktionen und Aufrufe, Ausdrücke
keine Variablen, Zuweisungen, Ablaufstrukturen
keine Seiteneffekte

Mächtige Programmierkonzepte durch Verwendung von
rekursiven Funktionen, Datenstrukturen, Funktionen
höherer Ordnung

Einzug in GPLs: Scala, Ruby, Python

Funktionale Programmierung

Grundkonzepte: Funktionen und Aufrufe, Ausdrücke
keine Variablen, Zuweisungen, Ablaufstrukturen
keine Seiteneffekte

Mächtige Programmierkonzepte durch Verwendung von
rekursiven Funktionen, Datenstrukturen, Funktionen
höherer Ordnung

Einzug in GPLs: Scala, Ruby, Python

```
def sum(f: Int => Int, a: Int, b: Int): Int =  
if (a > b) 0 else f(a) + sum(f, a + 1, b)
```

```
def powerOfTwo(x: Int): Int = if (x == 0) 1 else 2 * powerOfTwo(x - 1)
```

```
def sumPowersOfTwo(a: Int, b: Int): Int = sum(powerOfTwo, a, b)
```

Funktionale Programmierung

Binden von Ausdrücken an Namen

```
val message = "Hello World";
```

```
val x = message;
```

```
val message = "Goodbye GPS";
```

```
message;
```

```
x;
```

Funktionale Programmierung

Binden von Ausdrücken an Namen

```
val message = "Hello World";
```

```
val x = message;
```

```
val message = "Goodbye GPS";
```

```
message; // "Goodbye GPS"
```

```
x; // "Hello World"
```

Funktionale Programmierung

Binden von Ausdrücken an Namen

```
val name = Ausdruck;
```

```
val date = (14, "Juli", 2011);
```

```
date = (14.0, "Juli", 2011);
```

```
val twice = fn x => 2*x;
```

Funktionale Programmierung

Binden von Ausdrücken an Namen

```
val name = Ausdruck;
```

```
val date = (14, "Juli", 2011); // Tupel
```

```
date = (14.0, "Juli", 2011);
```

```
val twice = fn x => 2*x;
```

Funktionale Programmierung

Binden von Ausdrücken an Namen

```
val name = Ausdruck;
```

```
val date = (14, "Juli", 2011); // Tupel
```

```
date = (14.0, "Juli", 2011); // Fehler!
```

```
val twice = fn x => 2*x;
```

Funktionale Programmierung

Binden von Ausdrücken an Namen

```
val name = Ausdruck;
```

```
val date = (14, "Juli", 2011);
```

```
date = (14.0, "Juli", 2011);
```

```
val twice = fn x => 2*x;
```

```
twice 3;
```

```
twice (twice 3);
```

Funktionale Programmierung

Funktionen

```
val twice = fn x => 2*x; // Lambda-Funktion  
  
fun quad x = twice (twice x);
```

Funktionale Programmierung

Funktionen

```
val twice = fn x => 2*x; // Lambda-Funktion
```

```
fun quad x = twice (twice x);
```

```
fun quad x = twice twice x; //Fehler
```

Funktionale Programmierung

Funktionen

```
val twice = fn x => 2*x; // Lambda-Funktion
```

```
fun quad x = twice (twice x);
```

```
fun quad x = twice twice x; //Fehler
```

Funktionen mit mehreren Parametern

```
fun vonbis (m, n) =  
  if (m > n)  
    then nil  
  else  
    m::vonbis(m+1, n);
```

Funktionale Programmierung

Funktionen über Listen

```
fun prod numbers =  
  if numbers = nil  
  then 1  
  else (hd numbers) * prod (tl numbers);
```

hd, tl = eingebaute
Listenfunktionen

Funktionale Programmierung

Funktionen über Listen

```
fun prod numbers =  
  if numbers = nil  
  then 1  
  else (hd numbers) * prod (tl numbers);
```

prod in Musterschreibweise

```
fun prod nil = 1  
|   prod (h::t) = h* prod(t);
```

hd, tl = eingebaute
Listenfunktionen
:: Listenkonstruktor

Funktionale Programmierung

Funktionen höherer Ordnung
Funktionen als Parameter, first-class

```
fun foldl (f, a, nil) = a
| foldl(f, a, h::t) = foldl(f, f(a,h), t);
```

Funktionale Programmierung

Funktionen höherer Ordnung
Funktionen als Parameter, first-class

```
fun foldl (f, a, nil) = a
| foldl(f, a, h::t) = foldl(f, f(a,h), t);
Signatur: (('b * 'a) -> 'b * 'b * 'a list) -> 'b
```

Funktionale Programmierung

Funktionen höherer Ordnung
Funktionen als Parameter, first-class

```
fun foldl (f, a, nil) = a
| foldl(f, a, h::t) = foldl(f, f(a,h), t);
```

Anwendung:

```
fun product numbers = foldl(fn(a,h) => a*h, 1,
numbers);
```

Funktionale Programmierung

Zentral- vs. end-Rekursiv

```
fun Sum (nil) = 0
| Sum(h::t) = h + (Sum t);
```

```
Sum [1, 2, 3, 4]
```

Funktionale Programmierung

Zentral- vs. end-Rekursiv

```
fun Sum (nil) = 0
| Sum(h::t) = h + (Sum t);
```

Sum [1, 2, 3, 4]

[1, 2, 3, 4]

1+ ([2, 3, 4])

Funktionale Programmierung

Zentral- vs. end-Rekursiv

```
fun Sum (nil) = 0
| Sum(h::t) = h + (Sum t);
```

Sum [1, 2, 3, 4]

[1, 2, 3, 4]

1+ ([2, 3, 4])

1+ (2 +([3, 4]))

1+ (2 +(3+([4])))

Funktionale Programmierung

Zentral- vs. end-Rekursiv

```
fun Sum (nil) = 0
| Sum(h::t) = h + (Sum t);
```

Sum [1, 2, 3, 4]

[1, 2, 3, 4]

1+ ([2, 3, 4])

1+ (2 +([3, 4]))

1+ (2 +(3+([4])))

1+ (2 +(3+(4+[nil])))



Ergebnis
sukzessive nach
oben durchreichen

Operation jeweils
auf dem
Zwischenergebnis

zentral-rekursiv, da Additionen auf den
Zwischenergebnissen durchgeführt werden =>
Verwaltungsoverhead

Funktionale Programmierung

Zentral- vs. end-Rekursiv

```
fun Sum (nil) = 0
| Sum(h::t) = h + (Sum t);
```

```
Sum [1, 2, 3, 4]
```

[1, 2, 3, 4], 0

```
fun ASum (nil, a) = a
| ASum(h::t, a) = ASum(t, a+h);
```

```
ASum ([1, 2, 3, 4], 0)
```

Funktionale Programmierung

Zentral- vs. end-Rekursiv

```
fun Sum (nil) = 0
| Sum(h::t) = h + (Sum t);
```

```
Sum [1, 2, 3, 4]
```

[1, 2, 3, 4], 0

```
fun ASum (nil, a) = a
| ASum(h::t, a) = ASum(t, a+h);
```

[2, 3, 4], 1+0

```
ASum ([1, 2, 3, 4], 0)
```

Funktionale Programmierung

Zentral- vs. end-Rekursiv

```
fun Sum (nil) = 0
| Sum(h::t) = h + (Sum t);
```

```
Sum [1, 2, 3, 4]
```

[1, 2, 3, 4], 0

```
fun ASum (nil, a) = a
| ASum(h::t, a) = ASum(t, a+h);
```

[2, 3, 4], 1+0

[3, 4], 2+1

```
ASum ([1, 2, 3, 4], 0)
```

Funktionale Programmierung

Zentral- vs. end-Rekursiv

```
fun Sum (nil) = 0
| Sum(h::t) = h + (Sum t);
```

```
Sum [1, 2, 3, 4]
```

```
fun ASum (nil, a) = a
| ASum(h::t, a) = ASum(t, a+h);
```

```
ASum ([1, 2, 3, 4], 0)
```

[1, 2, 3, 4], 0

[2, 3, 4], 1+0

[3, 4], 2+1

[4], 3+3

Funktionale Programmierung

Zentral- vs. end-Rekursiv

```
fun Sum (nil) = 0
| Sum(h::t) = h + (Sum t);
```

```
Sum [1, 2, 3, 4]
```

```
fun ASum (nil, a) = a
| ASum(h::t, a) = ASum(t, a+h);
```

```
ASum ([1, 2, 3, 4], 0)
```

[1, 2, 3, 4], 0

[2, 3, 4], 1+0

[3, 4], 2+1

[4], 3+3

[], 4+6

Ergebnis
ohne Operation
darauf



Funktionale Programmierung

Rekursive Datenstrukturen: Bäume

```
datatype 'a tree = node of ('a tree * 'a * 'a tree) | treeNil;
```

Funktionale Programmierung

Rekursive Datenstrukturen: Bäume

```
datatype 'a tree = node of ('a tree * 'a * 'a tree) | treeNil;
```

New type names: =tree

```
datatype 'a tree =  
  ('a tree,  
   {con 'a node : ('a tree * 'a * 'a tree) -> 'a tree,  
     con 'a treeNil : 'a tree})  
con 'a node = fn : ('a tree * 'a * 'a tree) -> 'a tree  
con 'a treeNil = treeNil : 'a tree
```

Funktionale Programmierung

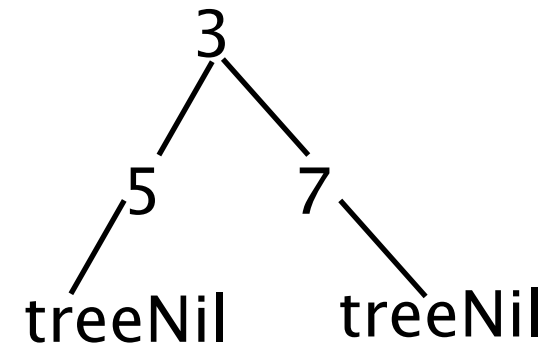
Rekursive Datenstrukturen: Bäume

```
datatype 'a tree = node of ('a tree * 'a * 'a tree) | treeNil;
```

New type names: =tree

```
datatype 'a tree =  
  ('a tree,  
   {con 'a node : ('a tree * 'a * 'a tree) -> 'a tree,  
     con 'a treeNil : 'a tree})  
con 'a node = fn : ('a tree * 'a * 'a tree) -> 'a tree  
con 'a treeNil = treeNil : 'a tree
```

```
val mytree = node(  
  node(treeNil, 5, treeNil),  
  3,  
  node(treeNil, 7, treeNil));
```



Funktionale Programmierung

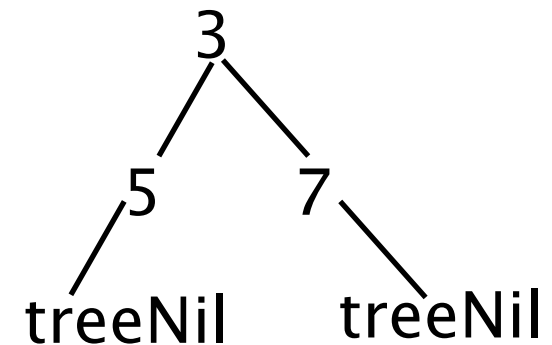
Rekursive Datenstrukturen: Bäume

```
datatype 'a tree = node of ('a tree * 'a * 'a tree) | treeNil;
```

New type names: =tree

```
datatype 'a tree =  
  ('a tree,  
   {con 'a node : ('a tree * 'a * 'a tree) -> 'a tree,  
     con 'a treeNil : 'a tree})  
con 'a node = fn : ('a tree * 'a * 'a tree) -> 'a tree  
con 'a treeNil = treeNil : 'a tree
```

```
val mytree = node(  
  node(treeNil, 5, treeNil),  
  3,  
  node(treeNil, 7, treeNil));
```



```
fun flatten (treeNil) = nil  
| flatten(node(l, v, r)) = (flatten l)@(v::(flatten r));
```

Funktionale Programmierung

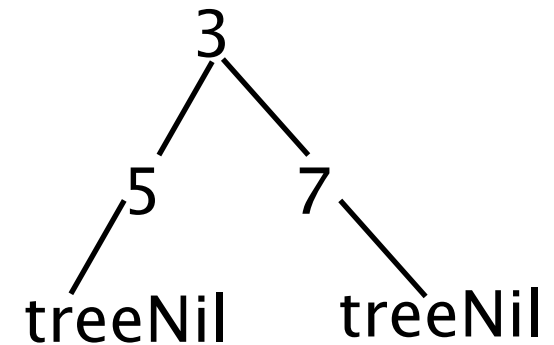
Rekursive Datenstrukturen: Bäume

```
datatype 'a tree = node of ('a tree * 'a * 'a tree) | treeNil;
```

New type names: =tree

```
datatype 'a tree =  
  ('a tree,  
   {con 'a node : ('a tree * 'a * 'a tree) -> 'a tree,  
     con 'a treeNil : 'a tree})  
con 'a node = fn : ('a tree * 'a * 'a tree) -> 'a tree  
con 'a treeNil = treeNil : 'a tree
```

```
val mytree = node(  
  node(treeNil, 5, treeNil),  
  3,  
  node(treeNil, 7, treeNil));
```



```
fun flatten (treeNil) = nil  
| flatten(node(l, v, r)) = (flatten l)@(v::(flatten r));
```

```
val 'a flatten = fn : 'a tree -> 'a list
```

Funktionale Programmierung

Rekursive Datenstrukturen: Bäume

```
fun flatten (treeNil) = nil
| flatten(node(l, v, r)) = (flatten l)@(v::(flatten r));

val 'a flatten = fn : 'a tree -> 'a list

fun quadtree(treeNil) = treeNil
| quadtree(node(l, v, r)) = node(quadtree(l), v*v, quadtree(r));
```

Funktionale Programmierung

Rekursive Datenstrukturen: Bäume

```
fun flatten (treeNil) = nil
| flatten(node(l, v, r)) = (flatten l)@(v::(flatten r));

val 'a flatten = fn : 'a tree -> 'a list

fun quadtree(treeNil) = treeNil
| quadtree(node(l, v, r)) = node(quadtree(l), v*v, quadtree(r));

> val quadtree = fn : int tree -> int tree
```

Funktionale Programmierung

Rekursive Datenstrukturen: Bäume

```
fun flatten (treeNil) = nil
| flatten(node(l, v, r)) = (flatten l)@(v::(flatten r));

val 'a flatten = fn : 'a tree -> 'a list

fun quadtree(treeNil) = treeNil
| quadtree(node(l, v, r)) = node(quadtree(l), v*v, quadtree(r));

> val quadtree = fn : int tree -> int tree

flatten(quadtree(mytree));

> val it = [25, 9, 49] : int list
```

Funktionale Programmierung

Rekursive Datenstrukturen: Bäume

```
fun flatten (treeNil) = nil
| flatten(node(l, v, r)) = (flatten l)@(v::(flatten r));

val 'a flatten = fn : 'a tree -> 'a list

fun quadtree(treeNil) = treeNil
| quadtree(node(l, v, r)) = node(quadtree(l), v*v, quadtree(r));

> val quadtree = fn : int tree -> int tree

flatten(quadtree(mytree));

> val it = [25, 9, 49] : int list

fun traverseTree(treeNil, f) = treeNil
| traverseTree(node(l,v,r), f) = node(traverseTree(l, f), f v,
traverseTree(r, f));
```